Modelling and Verification of Dynamic Role-Based Access Control

Inna Vistbakka[1(✉)] and Elena Troubitsyna[1,2]

[1] Åbo Akademi University, Turku, Finland
inna.vistbakka@abo.fi
[2] KTH, Stockholm, Sweden
elenatro@kth.se

Abstract. Controlling access to resources is essential for ensuring correctness of system functioning. Role-Based Access Control (RBAC) is a popular authorisation model that regulates the user's rights to manage system resources based on the user's role. In this paper, we extend the traditional static approach to defining RBAC and propose as well as formalise a dynamic RBAC model. It allows a designer to explicitly define the dependencies between the system states and permissions to access and modify system resources. To facilitate a systematic description and verification of the dynamic access rights, we propose a contract-based approach and then we demonstrate how to model and verify dynamic RBAC in Event-B. The approach is illustrated by a case study – a reporting management system.

1 Introduction

Modern software systems become increasingly resource-intensive. It is essential to guarantee that the authorised users have an access to the eligible resources and the resources are protected from an access by the unauthorised users. This aspect of the system behaviour is addressed by the access control policy.

Role-Based Access Control (RBAC) [4] is a widely used access control model. It regulates users' access to computer resources based on their role in an organisation. The standard RBAC framework adopts a static, state-independent approach to define the access rights to the system resources. However, it is often insufficient for correct implementation of the desired system functionality and should be augmented with the dynamic, i.e., a state-dependant view on the access control.

In this paper, we propose a dynamic RBAC model, which allows a designer to explicitly define the rights to access a certain resource based on the resource state and the system workflow. We formalise the dynamic RBAC and propose a systematic contract-based approach to defining the rights to access the system resources. We rely on the design-by-contract approach [8] to explicitly define the dynamic access rights for each role over resource. Moreover, we propose an approach that allow us to verify the consistency of the desired system workflow,

M. F. Atig et al. (Eds.): VECoS 2018, LNCS 11181, pp. 48–63, 2018.
https://doi.org/10.1007/978-3-030-00359-3_4

described by the scenarios, with the static and dynamic RBAC constraints. The workflow is described using UML use case and activity diagrams, which serve as a middle-hand between the textual requirements description and their formal Event-B model.

Event-B [2] is a state-based formalism for the correct-by-construction system development. It allows us to specify both dynamic and static aspects of system behaviour. The dynamic behaviour, defined by the events, models the workflow scenarios, which we want to analyse. The static component of the specification models the interdependencies between the roles, resources and the users. The Rodin platform and Pro-B plug-in [10,14] allow us to automate the verification of consistency between the dynamic RBAC and the desired system workflow. The approach is illustrated by a case study – a reporting management system.

2 From Static to Dynamic RBAC

RBAC: Basic Concepts. Role-Based Access Control (RBAC) [4] is one of the main mechanisms for ensuring data integrity in a wide range of computer-based systems. The authorisation model defined by RBAC regulates users' access to computer resources based on their role in an organisation.

RBAC is built around the notions of users, roles, rights and protected system resources. A *resource* is an entity, e.g., data, access to which should be controlled. A user can access a resource based on an assigned role, where a role is usually seen as a job function performed by a user within an organisation. In their turn, rights define the specific actions that can be applied to the resources. RBAC can be defined as a table that relates roles with the allowed rights over the resources. RBAC (depicted in Fig. 1) has the following elements:

- `USERS` is a set of users;
- `ROLES` is a set of available user roles;
- `RESOURSES` is a set of protected system resources;
- `RIGHTS` is a set of all possible rights over the resources;
- `PERMISSIONS` is a set of permissions over the resources.

Moreover, `US_ASSIGN` defines a *user assignment* to roles, while `RO_PERM` is *permission assignment* to roles. Next we discuss all these notions in details.

Let $\texttt{USERS} = \{\texttt{u}_1, \texttt{u}_2, ..., \texttt{u}_\texttt{n}\}$ be a set of users. In general, a concept of a user may stand for a person in the organisation, an administrative entity or a non-person entity, such as a computing (sub)system. A user can access a resource based on the assigned role.

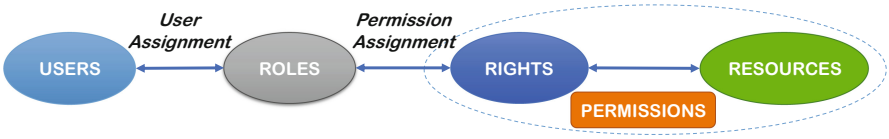

Fig. 1. RBAC structure

Let $\texttt{ROLES} = \{\texttt{r}_1, \texttt{r}_2, ..., \texttt{r}_\texttt{m}\}$ be a set of possible user roles within the system. A role is usually seen as a job function performed by a user within an organisation. For example, often, a role is used to indicate the job-related access rights to a resource.

The protected system resources are denoted by the set $\texttt{RESOURCES} = \{\texttt{re}_1, \texttt{re}_2, ..., \texttt{re}_\texttt{k}\}$. The notion of the resource depends on the system, i.e., it can denote OS files or directories; data base columns, rows or tables; disk space or just simple lock mechanisms.

Let $\texttt{RIGHTS} = \{\texttt{ri}_1, \texttt{ri}_2, ..., \texttt{ri}_1\}$ be a set of possible rights over the system resource. Rights are defined as specific manipulations that can be performed with the resources. For example, for a resource database, the access rights can be *Update, Insert, Append, Delete*; for a resource file – *Create, View, Print.*

A user can access resources based on the assigned roles. A user authorisation list – *user assignment* – can be defined as the mapping between users and roles:

$$\texttt{US\_ASSIGN} : \texttt{USERS} \rightarrow \mathbb{P}(\texttt{ROLES}),$$

which assigns to a given user a set of possible roles. A user can play (i.e., be mapped to) a number of roles, and a role can have many users. The notation $\mathbb{P}(\texttt{ROLES})$ stands for the powerset (set of all subsets) type over elements of the type ROLES.

Static RBAC. Access control in RBAC is realised in terms of (static) *permissions*. A permission is an ability of a holder of a permission to perform some action(s) in the system. To formally define all possible permissions, we introduce the relation PERMISSIONS as follows:

$$\texttt{PERMISSIONS} : \texttt{RESOURCES} \leftrightarrow \texttt{RIGHTS}$$

It describes relationships between a certain system resource and the rights that can be applied to it.

Permission assignments to a role are defined based on the job authority and responsibilities within the job function. To formally define permissions that are provided by the system to the different user roles, we define the function RO_PERM that maps each user role to a set of allowed rights over the resources:

$$\texttt{RO\_PERM} : \texttt{ROLES} \rightarrow \mathbb{P}(\texttt{PERMISSIONS}).$$

In the paper, we make a distinction between rights and operations. The operations (or *use cases*) define the specific tasks, which a user may perform in the system. Therefore, an "operation" is a more general concept than a "right" and designates specific basic rights which are invoked by a user. Let us consider a resource "personal profile page" – page, which is typically created for each employee in an organisation. The set of the access rights for this resource includes *Create, Delete, Read, Write.* An examples of user's operation within a system can be "View Personal Profile", "Edit Personal Profile", etc. To be successfully executed, the operation "View Personal Profile" requires the *Read* right, while the "Edit Personal Profile" operation requires both *Read* and *Write* rights.

Usually RBAC gives a static view on the access rights associated with each role, i.e., it defines the permissions to manipulate certain resources "in general", i.e., without referring to the system state. Therefore, rights define the necessary conditions for an operation to be executed. However, we argue, that these conditions are insufficient for a correct implementation of the intended system functionality. For instance, assume that a user with a specific role `User` has *Read* and *Write* rights to the personal profile page `page`, where a user with the role `Admin` has *Read*, *Write* as well as *Create* and *Delete* rights to the resource `page`. Even though `User` has rights to *Read* and *Write* the profile page, s/he cannot use them if `Admin` has not created the web-page before using his/her right *Create* or has already deleted it using *Delete* right.

It is easy to see that the access rights depend not only on the role but also on the state of the resource. Therefore, the static view on RBAC should be complemented with an explicit definition of the *dynamic state-dependant* conditions.

Dynamic RBAC. Let us now discuss a formalisation of the dynamic view on RBAC. Each resource can be characterised by its state, i.e., we can introduce the set $\texttt{STATES} = \{\texttt{st}_1, ..., \texttt{st}_\texttt{j}\}$ defining all possible states of the resources. Then we can define *dynamic (state-dependant) permissions* as the following function:

$$\texttt{DYN\_PERM} : \texttt{RESOURCES} \times \texttt{STATES} \rightarrow \mathbb{P}(\texttt{RIGHTS}).$$

For each resource and its specific state, `DYN_PERM` returns access rights applicable to the resource in each of its states. Let us note, that `DYN_PERM` is defined for all allowed access rights that can be applied to the resources. Then *dynamic role permissions* can be defined as the function `DYN_RO_PERM`:

$$\texttt{DYN\_RO\_PERM} : \texttt{ROLES} \rightarrow \mathbb{P}(\texttt{DYN\_PERM}).$$

Essentially, it maps the assigned dynamic permissions to the roles.

Let us now return to our personal profile page example. Assume that the resource `page` can be in three states: *null* (before it is created), *locked* (after it is deleted) or *unlocked*. Then, when `page` is in the state *null*, `User` has no rights over this resource. However, when `page` is in the state *unlocked*, `User` has *Read* and *Write* rights, and when `page` is in the state *locked*, `User` role has *Read* right.

The dynamic and static views on RBAC are intrinsically interdependent. The permissions defined by the static and dynamic constraints constitute the necessary and sufficient constraints the user has over the operations execution. In the next section, we discuss how to verifying these conditions using the design-by-contract approach.

3 Reasoning About Dynamic RBAC Using Contracts

The dynamic view on RBAC, advocated in this paper, aims at defining conditions enabling a successful execution of an operation with respect to both –

static access rights defined by RBAC and system dynamics defined by its workflow. Typically, the workflow is described by the *scenarios*. A scenario defines a sequence of operations – *use cases* – that should be performed over the resources to implement the desired functionality. A scenario consists of individual steps that combine the operations executed over the resources in a certain order.

Usually a scenario involves a single or multiple actors (users) that perform the operations over the resources. The users performing the operations in a scenario must have all the permissions required to complete every single step of a given scenario. Thus we should verify consistency between the defined RBAC and the control flow implemented by the desired scenarios. For each operation in the scenario, we define the correctness conditions as the *contract for operation*. We follow the design-by-contract approach [8], i.e., define each contract as a combination of a *precondition* (the conditions on the operational input) and a *postcondition* (conditions to be satisfied as a result of the operation execution).

A Concept of an Operation Contract. Let *OPERATIONS* be a set representing all possible operations within a system execution. Each operation represents an interaction of a user with the system. We assume that the state of a system is represented by a collection of variables denoted as v. Then the user operations result in changing the system state.

For each operation we define a **pre**- and **post**-condition pair. Figure 2 presents a generic from of an operation structure definition. An operation $oper_i$ might have parameters p_i that are defined in the **params** clause. The **pre**-clause defines the assumptions about the state of the system before the execution of the operation. The **post**-clause defines the state of the system after the completion of the operation. Here postconditions describe the actual changes in the state of the resource. The operation as such is a state transition resulting in the change of the variables values from v to v'.

A precondition represents the static and dynamic constraints that should be satisfied by each operation. If precondition of an operation is not satisfied then the scenario containing it is deadlocked indicating an inconsistency between the formulated constrains and the desired workflow.

Defining Consistency Conditions. Let us now investigate how to use contracts to derive consistency conditions. We start by introducing the function *ScenarioSeq* such that

$$ScenarioSeq \in SCENARIOS \rightarrow \texttt{seq}(OPERATIONS),$$

Operation $oper_i$	// <name of an operation>
params p_i	// <list of parameters>
pre $Pre_i(p_i, v)$	// <list of predicates>
post $Post_i(p_i, v, v')$	// <list of predicates>
end	

Fig. 2. General structure of an operation contract

where `seq` is a sequence constructor to represent composite steps within a scenario. Here *SCENARIOS* is a set of scenario ids.

Lets consider a scenario S, where $S \in SCENARIOS$. We say that a scenario S is *executable* if the final state of a scenario S is reachable from its initial state. Since a scenario is defined as a sequence of the corresponding operations, and the operations, in their turn, are defined as state transitions, we can define a scenario execution as follows:

$$\sigma^S_{init} \rightsquigarrow^S \sigma^S_{fin}, \tag{1}$$

where σ^S_{init} and σ^S_{fin} denote the initial and final states of a scenario S respectively.

Let a scenario S be a sequence consisting of m operations:

$$ScenarioSeq(S) = [oper_1, oper_2, ..., oper_m], \tag{2}$$

We use the definition of an operation contract to explicitly formulate the consistency property of a scenario control flow. A scenario S is executable (i.e., a scenario control flow is consistent) if the following properties are hold:

$$Pre(oper_{i+1}) \subseteq Post(oper_i), i = 1...m-1, \tag{3}$$

$$Pre(oper_1) \subseteq \sigma^S_{init}, \tag{4}$$

$$\sigma^S_{fin} \subseteq Post(oper_m) \tag{5}$$

Essentially, these properties require that all sequences of the scenario steps are enabled. Property (3) requires that any next operation should be *enabled* by the previously executed operation. Property (4) describes the consistency conditions imposed on the initial state of a scenario, i.e., verifies that the first operation is enabled. Property (5) requires for the final state of the scenario to be a subset of the states in which the last operation in the scenario terminates.

The above definitions formalise the constraints that should be verified to ensure that the operations of a given scenario can be executed in the desired order. If any of the conditions is violated then an inconsistency is detected, which should be eradicated either by inspecting the requirements or their formalisation. Next we will discuss how to use the proposed formalisation in the context of dynamic RBAC.

Operation Implementation Under RBAC. The generic structure of an operation description is given in Fig. 2. In the RBAC context, an operation defines user action over a system resource. Upon an operation execution, the state of the resource might be changed. Consequently, it might result in changing the (dynamic) access rights for a particular role over resources. Thus, in the context of RBAC, we can define an operation as shown in Fig. 3.

Below we give an explanation of each clause:

- **params** *clause.* The user operation over the system resource has following parameters: a user ***us***, a user role ***ro*** and a resource ***res***.
- **pre** *clause.* Predicates over

Operation $oper_i$
params *us, res, ro*
pre Resource_State(*res*) = *state*
US_ASSIGN(*us*) = *ro*
rights(ro) ∈ DY_RO_PERM(ro)(*res*, *state*)
post Resource_State(res) = $state'$
$rights'$ *[roles]* = DY_RO_PERM*[roles]*(*res*, $state'$)
end

Fig. 3. A generic operation implementation for RBAC

- a current *state* of the resource ***res***;
- required access *rights* of the role **ro** over the resource ***res*** to perform the operation.

– **post** *clause*. Predicates over
- modified *state* of a resource ***res***;
- revised access *rights* for all ***roles*** over the resource ***res***.

The precondition aims at verifying that the resource is in the correct state before the operation execution, the user has a role that makes him/her eligible for executing this operation, and the operation can be executed with respect to the current resource state and the role. The postcondition postulates that the state of the resource might change as well as the dynamic rights for the system roles. Let us observe, that the input parameter role ***ro*** does not change as a result of the operation execution. However, it should be defined since the same operation would typically have different contracts for different roles.

In this section, we have defined the conditions which should be verified to ensure consistency of desired scenarios and static and dynamic RBAC constraints. To automate the proposed approach, we propose to use Event-B framework. In the next section, we give its brief overview.

4 Background: Event-B and ProB

The Event-B formalism [2] is a state-based formal approach that promotes the correct-by-construction system development and formal verification by theorem proving. In Event-B, a system model is specified as an *abstract state machine* [2]. An abstract state machine encapsulates the model state, represented as a collection of variables, and defines state operations, i.e., it describes the dynamic system behaviour. Types of variables and other properties are defined in the *Invariants* clause. A machine also has an accompanying component, called *context*, which includes user-defined sets, constants and their properties given as model axioms.

The dynamic behaviour of the system is defined by a collection of atomic *events*. Generally, an event has the following form:

$$e \;\widehat{=}\; \textbf{any}\ a\ \textbf{where}\ G_e\ \textbf{then}\ R_e\ \textbf{end},$$

where e is the event's name, a is the list of local variables, G_e is the event *guard*, and R_e is the event action. The *guard* G_e is a predicate over the local variables of the event and the state variables of the system. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment over the system variables. The guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

The system behaviour in Event-B is modelled by a set of events. We can transform this representation into the pre- postcondition format, as we did in [16] and then establish the correspondence between the definitions of an operation contract and an Event-B event. To perform it we can rely on our previous work presented, e.g., in [6].

Refinement in Event-B. Event-B employs a top-down refinement-based approach to system development. Development typically starts from an abstract specification that nondeterministically models most essential functional requirements. In a sequence of refinement steps, we gradually reduce nondeterminism and introduce detailed design decisions. The consistency of Event-B models, i.e., verification of well-formedness, invariant preservation as well as correctness of refinement steps, is demonstrated by proving the relevant verification theorems – proof obligations [2].

Tool Support for Development and Model Checking. Modelling, refinement and verification in Event-B is supported by an automated tool – Rodin platform [14]. The platform provides the designers with an integrated modelling environment, supporting automatic generation and proving of the proof obligations. Moreover, various Rodin extensions allow the modeller to transform models from one representation to another. They also give access to various verification engines (theorem provers, model checkers, SMT solvers).

For instance, the ProB extension [10] of Rodin supports automated consistency checking of Event-B machines via model checking, constraint based checking, and animation. ProB supports analysis of liveness properties (expressed in linear or computational tree logic (LTL/CTL)), invariant violations as well as the absence of deadlocks. A model checker systematically explores the state space, looking for various errors in the model under consideration [7].

In this paper, we argue that modelling with Event-B provides us with a suitable verification dynamic policies of RBAC.

5 Verification of System Scenarios Under Dynamic RBAC

In Sects. 2 and 3 we discussed the dynamic extension of RBAC and defined the conditions for verifying scenario consistency. In this section, we propose a formal Event-B based approach that implements these ideas to identify possible scenarios violating consistency. Our approach uses the graphical modelling in UML as the front-end. Graphical models are used to describe the general structure

of the system and its scenarios. They serve as a middle-hand between the textual requirements description and formal specification. The proposed approach is shown in Fig. 4. It consists of the following six steps:

Step 1: *Define RBAC model.* Define the system roles and operations over the system resources for each role. Represent the actors as roles, the operations as use cases and create the UML use case diagram of a system. Create an activity diagram representing the intended system workflow.

Step 2: *Define/modify operation implementations.* Define all possible states of the system resources and create a state diagram representing how execution of each actor's operation changes the state of the resources. Then using the created state diagram, create an abstract specification in Event-B that defines the resource states and the corresponding state transitions. Each defined state transition should correspond to a realisation of the user operation.

Next, for each operation, define (or modify) its contract. Represent all the required elements: resource state, a role, required basic access rights for a role to perform an operation and the id of the accessed resource. Incorporate into the Event-B model the defined contracts by specifying the Event-B events.

Step 3: *Compose a scenario and check it for consistency.* Compose/modify a scenario over the operations defined in the Step 2. All the dynamic characteristics of a system are formulated in terms of the model variables and the required properties as the model invariants. All the static system properties are defined in the model context (e.g., a sequence of evolved scenario steps). Then simulate execution of a chosen scenario in Event-B and model check this model in ProB looking for inconsistencies in its execution. Any violations of the control flow consistency conditions (3)–(5) lead to deadlocking the model, which in turn indicate such inconsistencies in the operation definitions.

Step 4: *Scenario analysis.* If a deadlock in the previous Step 3 is found for a certain scenario, then analyse the operations involved in the scenario execution. The purpose of a such analysis is to come up with one or several recommendations for modification of the operation implementation. Then return to the Step 2 for necessary modifications of one or several operations.

Step 5: *Storing a valid scenario.* A checked valid scenario is stored as the corresponding command sequence in the Event-B context. Return back to Step 3 until a scenario model is complete.

The resulting Event-B model (specification) can be used as an input for the next system development steps. Event-B specification can be refined further to introduce detailed requirements representation. Since the consequent refinement steps depend on the nature of the system to be developed, we omit their consideration in this paper.

In the next section, we illustrate the proposed approach by an example.

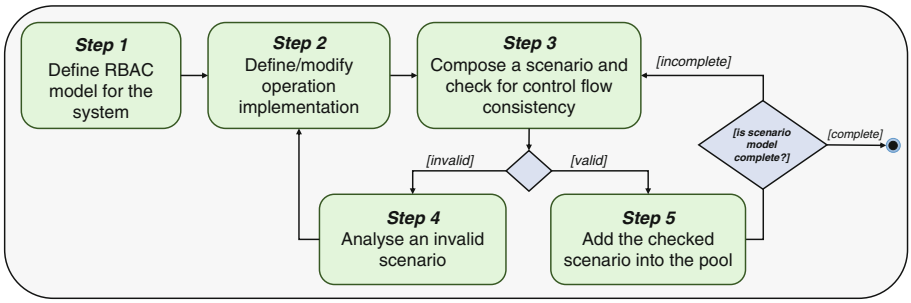

Fig. 4. Steps of the approach

6 Formal Modelling of a Reporting Management System

The Reporting Management System (RMS) is used by different employees in an organisation to send periodic (e.g., monthly) work reports. Below we summarise the access control-related requirements:

- *There are three roles in the system* – employee, controller and administrator;
- *Functionality associated with the roles*:
  - An employee can create a new report, modify an existing one or delete a non-approved report, as well as submit a report to its controller;
  - A controller can read the submitted report received from one of the associated employees, and can either approve or disapprove it;
  - An administrator has an access to all the reports of all her/his associated controllers, and it is her/his responsibility to register the reports approved by the controllers.
- *Report access policies*:
  - Until a report is submitted, the employee can modify or delete it.
  - As soon as a report is submitted, it cannot be altered or deleted by the employee any longer.
  - Upon the controller's approval, the report is registered by the administrator.
  - In case of disapproval, the report is returned back to the employee and can be further modified or deleted.

Let us note that each actor operation requires certain basic access rights. For instance, an employee, to execute `Modify Report` operation, should have *Read* and *Write* access rights to the report file. In its turn, the employee's supervisor – controller – should have *Read* and *Write* access rights to the same file to execute `Approve Report` operation. However, as soon as an employee submits a report to a controller, she/he can have only *Read* access right to the report file.

System Modelling and Verification. To specify and verify RMS, we follow the steps described in Sect. 5:

Step 1. We identify the main roles and their operations and create the use case diagram as shown in Fig. 5(a). It shows the actors, their roles in the system and also their possible interactions with the system. Also we create the activity diagram presented in Fig. 5(b) to describe the workflow associated with the defined functions.

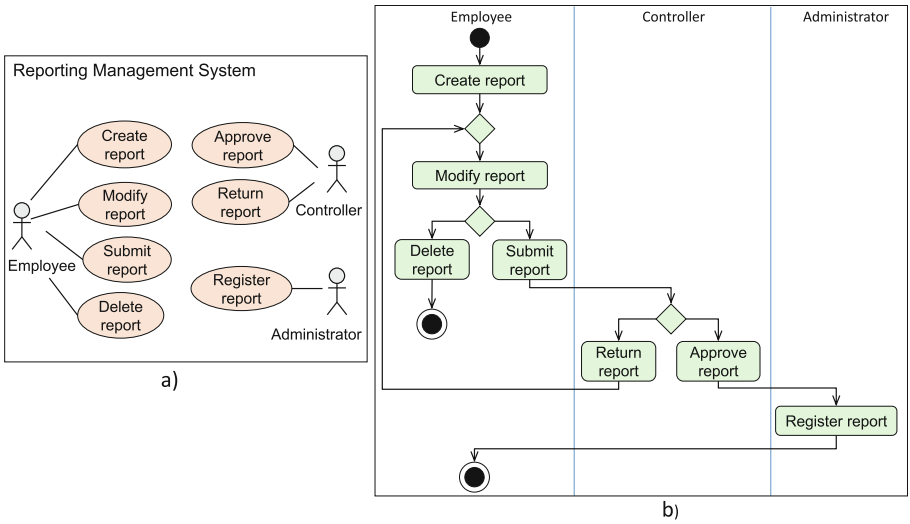

Fig. 5. Reporting management system: (a) Use case diagram, (b) Activity diagram

Step 2. Each actor's function changes the state of a certain report. Hence, the overall behaviour of the system, for each particular report, can be considered as a set of transitions between all the possible states of the report. The corresponding state diagram is represented in Fig. 6.

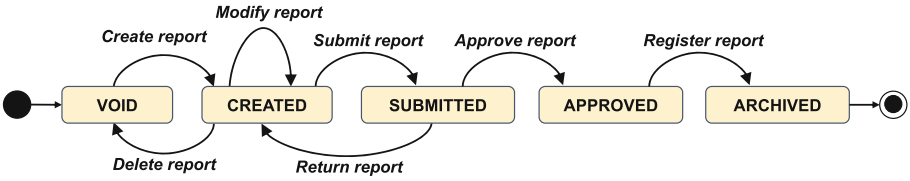

Fig. 6. State diagram – reporting management system

Then, we use the state diagram and create an abstract specification in Event-B that defines the state of reports and the corresponding state transitions. To represent a current state for each report we define a function *report_state*:

$$report\_state \in REPORTS \rightarrow STATES.$$

Initially each report has the state *VOID*. The actual report creation is modelled by the event CreateReport that changes the state of a single report *rp* to *CREATED*. Then the events ModifyReport, DeleteReport and SubmitReport become enabled. When the report is submitted, its state changes to *SUBMITTED*. Upon report approval, its state is changed to *APPROVED*, otherwise, if the report is rejected, it returns back to the state *CREATED*. Finally, once the administrator registers already approved report, the report goes to its final state *ARCHIVED*.

Next we link each role with the set of operations that correspond to it. Moreover, for each role, we will define the required basic access rights – *Create, Read, Write, Delete* – modelled as *C, R, W, D* values, respectively.

To specify dynamic permissions for the introduced roles, we define a variable *dPerm* with the following properties:

$$dPerm \in ROLES \times REPORTS \rightarrow \mathbb{P}(RIGHTS),$$
$$\forall\, r \in REPORTS \cdot dPerm(\texttt{Employee}, r) \subseteq \{C, R, W, D\} \wedge$$
$$dPerm(\texttt{Controller}, r) \subseteq \{R, W\} \;\wedge\; dPerm(\texttt{Administrator}, r) \subseteq \{R, W\}.$$

The variable *dPerm* is a function that assigns to each role and a report a set of possible access rights that can be associated with the role.

Obviously, for each role, the set of available access rights to a report depends on the current state of this report. For instance, a controller can have *Read* (R) and *Write* (W) rights only over the submitted reports. Moreover, when the report that has been submitted for approval, it cannot be further modified by the employee until the end of the approval period. Therefore, during the approval period, the employee has only *Read* right to this particular report. Hence, we should restrict the set of enabled rights depending on the report's state. In the corresponding Event-B events which model the change of a report state, the values of role permissions will be updated. For brevity, we omit showing the whole Event-B specification – its excerpt is presented in Fig. 7. For more details, please see our previous work [20].

Let us note that the variables *dPerm* and *report_state* together represent the *dynamic role permissions* RO_DYN_PERM discussed in Sect. 2. We use these two variables instead of one just to avoid nested data structures (function of function) in Event-B specification.

Step 3: Scenario Verification. In this step, we analyse the desired system scenarios associated with RMS. For example, we consider a simple scenario as the following chain of operations performed by an employee and a controller:

CreateReport → ModifyReport → SubmitReport → ReturnReport → ModifyReport

We represent this chain as the corresponding command sequence and define it in the context. Then, in the machine part of the Event-B specification of RMS, we simulate the scenario execution by accumulating the information about the sequence of the corresponding scenario steps in *Scenario*. To implement it, we define a number of events – Start, Next, Finish – that simulate the scenario execution (see Fig. 8). The sequence of operations is built by starting from the first operation and simulating the execution sequenceleading to the last operation.

The scenario execution process is completed when the last command of the scenario is executed.

We formulate the invariant property *finish=TRUE⇒ CurScenario=Scenario* stating that if the scenario execution has been completed then the scenario contains all the steps (i.e., is equal to the executed steps). In the case, when the resulting command sequence does not match to the required sequence (*CurScenario*), a violation is found by model checking. Consequently, a found scenario sequence becomes an input for Step 4.

Step 4: Analysis and Operation modifications. In our example, we have found a deadlock – the scenario execution deadlocks on the execution of the event ReturnReport. We analysed the operation and discovered that in the implementation of ReturnReport the access rights for `Employee` upon the operation execution are set to R (*Read* right). However, the operation ModifyReport requires for a role `Employee` to have *R, W, D* access rights (*Read*, *Write* and *Delete*, respectively) to the report file. As a result, we modify the operation ReturnReport and check again this scenario for consistency.

Step 5: Storing a scenario. The checked scenario from *CurScenario* is then added to the set of checked scenarios *CheckedScenarios*. Then we repeat phases Step 3–5 until we verify all the scenarios in the desired system workflow represented in the activity diagram.

As a result of the described process, we arrive at an Event-B model of RMS. We specify and verify dynamic access control via allowed rights over the resources according to the system policies.

7 Related Work and Conclusions

Recently the problem of modelling and analysing the access control policies has attracted a significant research attention. Milhau et al. [9] have proposed

Machine RMS_abs
Events...
CreateReport $\widehat{=}$
 any rep, u
 where $report\_state(rep) = VOID$
  $US\_ASSIGN(u) = \texttt{Employee}$
  $C \in dPerm(\texttt{Employee}, rep)$
 then
  $report\_state(rep) := CREATED$
  $dPerm(\texttt{Employee}, rep) := \{R, W, D\}$
 end
ModifyReport $\widehat{=}$
 any rep, u
 where $report\_state(rep) = CREATED$
  $US\_ASSIGN(u) = \texttt{Employee}$
  $\{R, W, D\} \subseteq dPerm(\texttt{Employee}, rep)$
 then
  $skip$
 end
SubmitReport $\widehat{=}$
 any rep, u
 where $report\_state(rep) = CREATED$
  $US\_ASSIGN(u) = \texttt{Employee}$
  $R \in dPerm(\texttt{Employee}, rep)$
 then
  $report\_state(rep) := SUBMITTED$
  $dPerm := dPerm \Leftarrow (\{\texttt{Employee} \mapsto rep \mapsto \{R\}\} \cup \{\texttt{Controller} \mapsto rep \mapsto \{R, W\}\})$
 end
ReturnReport $\widehat{=}$
 any rep, u
 where $report\_state(rep) = SUBMITTED$
  $US\_ASSIGN(u) = \texttt{Controller}$
  $R \in dPerm(\texttt{Controller}, rep)$
 then
  $report\_state(rep) := CREATED$
  $dPerm := dPerm \Leftarrow (\{\texttt{Employee} \mapsto rep \mapsto \{R\}\} \cup \{\texttt{Controller} \mapsto rep \mapsto \varnothing\})$
 end
...

Fig. 7. Event-B specification of RMS (with possible inconsistencies)

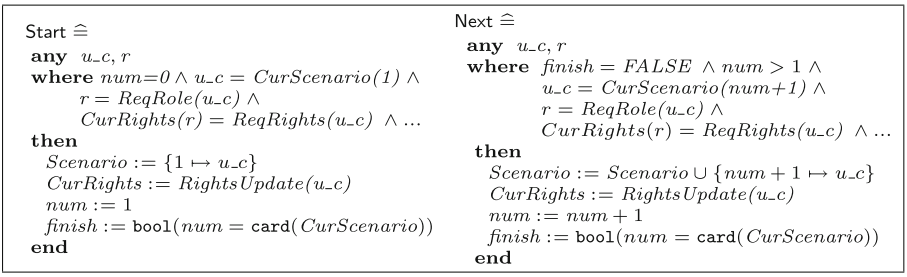

Fig. 8. Step 3: some events of the formalisation

a methodology for specifying access control policies using a family of graphical frameworks and translating them into the B. The main aim of the work has been to formally specify an access control filter that actually regulates access control to the data. In this work, the dynamics is mainly considered with respect to the operation execution order, while, in our work, the dynamic view on the access policies depends on the system state, in particular, on the state of a resource.

The basic RBAC model has been extended in a variety of ways [1,5,13]. The problem of spatio-temporal RBAC model is discussed in [1]. The authors considered role-based access control policies under time and location constraints. Moreover, they demonstrated how the proposed model can be represented and analysed using UML and OCL. Ray et al. [13] proposed location-aware RBAC model that incorporates location constraints in user-role activation. In our work we consider dynamic, state-dependent constraints within the access control model.

A number of works uses UML and OCL based domain specific language to design and validate the access control model. For instance, in the work [15] UML is used to describe security properties. In contrast to our work, here the authors transform UML models to Alloy for analysis purpose. A domain-specific language for modelling RBAC and translating graphical models in Event-B was proposed in [19].

Verification of behaviour aspects of software models defined using the design-by-contract approach has been discussed, e.g., in [3]. The goal of this work has been to detect the defects in the definition of the operations. Formal verification has been performed over the declarations of the operations in the UML/OCL models. In contrast, in our work, the defined operational contracts are used to model the scenario execution sequences and formulate the consistency properties. A contract-based approach to modelling and verification of RBAC for cloud was proposed in [11]. An approach to integrating UML modelling and Event-B to reason about behaviour and properties of web-services was proposed in [12].

A data-flow oriented approach to graphical and formal modelling has been proposed in [17,18,21]. These works use the graphical modelling to represent system architecture and the data flow. The diagrams are translated into Event-B, to verify the impact of security attacks on the invariant system properties.

In this paper, we have done two main research contributions. Firstly, we have defined a formal model of dynamic RBAC and proposed a contract-based approach to verification of consistency of scenarios with respect to the static and dynamic RBAC constraints. Secondly, we have proposed an integrated approach incorporating verification of dynamic RBAC into Event-B. In our approach, graphical models are used as a middle hand between the textual requirements description and a formal model. They help a designer to identify the scenarios and define the system workflow.

In this paper, we have used a combination of proving and model checking to verify consistency between the scenarios and the constraints of dynamic RBAC. Event-B and the Rodin platform have offered us a suitable basis for the formalisation and automation of our approach. The provers have been used to verify correctness of the data structure definitions and the Pro-B model checker to find violations in the scenario models. Moreover, model animation has facilitated analysis of the scenarios as well as identifying the recommendations for operation implementation specifications. We have validated our approach by a case study – Reporting Management System. We believe that the proposed approach facilitates an analysis of complex access control policies.

As a future work, we are planing to consider more complex variants of dynamic RBAC. For instance, we will model the situations when several users can get simultaneous or partial access to some parts of a data resource depending on their roles and resource states. Moreover, we are planing to work on an extension of the proposed approach for modelling and verification of dynamic RBAC and formalise it as Event-B specification patterns.

References

1. Abdunabi, R., Al-Lail, M., Ray, I., France, R.B.: Specification, validation, and enforcement of a generalized spatio-temporal role-based access control model. IEEE Syst. J. **7**(3), 501–515 (2013)
2. Abrial, J.R.: Modeling in Event-B. Cambridge University Press, Cambridge (2010)
3. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL operation contracts. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 40–55. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00255-7_4
4. Ferraiolo, D.F., Sandhu, R.S., Gavrila, S.I., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. **4**(3), 224–274 (2001)
5. Fuchs, L., Pernul, G., Sandhu, R.S.: Roles in information security - a aurvey and classification of the research area. Comput. Secur. **30**(8), 748–769 (2011)
6. Laibinis, L., Troubitsyna, E.: A contract-based approach to ensuring component interoperability in Event-B. In: Petre, L., Sekerinski, E. (eds.) From Action Systems to Distributed Systems - The Refinement Approach, pp. 81–96. Chapman and Hall/CRC (2016)
7. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008)
8. Meyer, B.: Design by contract: the Eiffel method. Proc. Tools **26**, 446 (1998)

9. Milhau, J., Idani, A., Laleau, R., Labiadh, M., Ledru, Y., Frappier, M.: Combining UML, ASTD and B for the formal specification of an access control filter. ISSE **7**(4), 303–313 (2011)
10. ProB: Animator and Model Checker. https://www3.hhu.de/stups/prob/index.php/. Accessed 06 June 2018
11. Rauf, I., Troubitsyna, E.: Generating cloud monitors from models to secure clouds. In: DSN 2018. IEEE Computer Society (2018, in print)
12. Rauf, I., Vistbakka, I., Troubitsyna, E.: Formal verification of stateful services with REST APIs using Event-B. In: IEEE ICWS 2018. IEEE (2018, in print)
13. Ray, I., Kumar, M., Yu, L.: LRBAC: a location-aware role-based access control model. In: Bagchi, A., Atluri, V. (eds.) ICISS 2006. LNCS, vol. 4332, pp. 147–161. Springer, Heidelberg (2006). https://doi.org/10.1007/11961635_10
14. Rodin: Event-B platform. http://www.event-b.org/. Accessed 06 June 2018
15. Sun, W., France, R.B., Ray, I.: Rigorous analysis of UML access control policy models. In: POLICY 2011, pp. 9–16. IEEE Computer Society (2011)
16. Tarasyuk, A., Troubitsyna, E., Laibinis, L.: Integrating stochastic reasoning into Event-B development. Formal Asp. Comput. **27**(1), 53–77 (2015)
17. Troubitsyna, E., Laibinis, L., Pereverzeva, I., Kuismin, T., Ilic, D., Latvala, T.: Towards security-explicit formal modelling of safety-critical systems. In: Skavhaug, A., Guiochet, J., Bitsch, F. (eds.) SAFECOMP 2016. LNCS, vol. 9922, pp. 213–225. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45477-1_17
18. Troubitsyna, E., Vistbakka, I.: Deriving and formalising safety and security requirements for control systems. In: SAFECOMP 2018. LNCS. Springer, Cham (2018, in print)
19. Vistbakka, I., Barash, M., Troubitsyna, E.: Towards creating a DSL facilitating modelling of dynamic access control in Event-B. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 386–391. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_28
20. Vistbakka, I., Troubitsyna, E.: Towards integrated modelling of dynamic access control with UML and Event-B. In: IMPEX/FM&MDD 2017. EPTCS, vol. 271, pp. 105–116 (2018)
21. Vistbakka, I., Troubitsyna, E., Kuismin, T., Latvala, T.: Co-engineering safety and security in industrial control systems: a formal outlook. In: Romanovsky, A., Troubitsyna, E.A. (eds.) SERENE 2017. LNCS, vol. 10479, pp. 96–114. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65948-0_7